A Framework of Application Generator Design

Scott Thibault, Charles Consel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Framework of Application Generator Design

Scott Thibault, Charles Consel

N ° 3005

Octobre 1996

THÈME 2

ISSN 0249-6399

INRIA
RENNES

A Framework of Application Generator Design

Scott Thibault, Charles Consel

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 3005 — Octobre 1996 — 8 pages

Abstract: Application generators have been demonstrated as a successful approach to achieving software reuse and typically yields higher productivity gains than methods such as component-based reuse. Despite their advantages, industrial software developers are reluctant to adopt these methods due to the lack of tools for constructing generators.

This paper presents a framework for the development of application generators. This framework provides a structured design approach and automatic tools for design. The framework consists of a two level design process: The first level is the identification of operations that expresses the fundamental computations of the application domain. The second level is the design of a domain-specific language which allows one to express variations within a family of applications. The domain-specific language is implemented in terms of the operations defined by the first level. We show that the uniform application of partial evaluation enables automatic application generation from a micro-program to its implementation. This framework has been developed in the context of real applications in areas such as Internet services and digital television, and is being developed in conjunction with industrial partners.

Key-words: Domain-Specific Languages, Application Generators, Reuse, Partial Evaluation

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Un Cadre Général pour le Design de Générateurs d'Applications

Résumé : Les générateurs d'applications sont une approche très prometteuse pour la réutilisation logicielle. En particulier, ils apportent des gains de productivité supérieurs aux méthodes basées sur la réutilisation de composants logiciels. Cependant, malgré ses avantages, l'industrie du développement logiciel rechigne à adopter cette approche du fait de l'absence d'outils d'aide pour la construction de générateurs d'applications.

Ce rapport présente un cadre général pour le développement de tels générateurs. Nous décrivons comment structurer le design des générateurs d'applications et quels outils automatiques employer pour leur construction. Le premier niveau consiste à identifier le noyau des opérations fondamentales du domaine d'applications. Le second niveau est le design d'un langage dédié qui permet d'exprimer des variations à l'intérieur d'une famille d'applications. Un micro-programme dans ce langage dédié est implémenté à l'aide des opérations définies dans le premier niveau. Nous montrons par ailleurs comment l'utilisation généralisée de l'évaluation partielle permet la génération automatique d'une application, partant d'un micro-programme pour fournir finalement une implémentation.

Ce cadre général à été développé dans le contexte d'applications réelles dans des domaines comme les services sur Internet et la télévision digitale. Son développement se poursuit en collaboration avec des partenaires industriels.

Mots-clé : Langages Dédiés, Générateur d'Applications, Réutilisation, Évaluation Partielle

1 Introduction

Our framework for the development of application generators is structured into two levels. The first level is based on well established ideas of generic components [14, 4], which have desirable characteristics for reuse and structured design. This level of the framework consists of the definition of an abstract machine, where operations of the abstract machine can be viewed as generic components. Abstract machines are a well understood concept from programming languages [12] that have several advantages discussed in section 4, such as providing an abstraction to multiple implementations.

Given the definition of an abstract machine, which defines a collection of operations suitable for building applications within a domain, the next step is to provide a method of composing operations to form an application. Composing these operations through the use of a glue language can be cumbersome and similar applications will often share a similar structure. Our approach consists of using a domain-specific language, as an interface to the abstract machine. Thus, the second level of our framework is the definition of a micro-language. The micro-language represents an interface to the abstract machine that, among other things, allows reuse among similar applications and can be used by non-programmers.

The result of this two-level framework is a structured approach to the development of an application generator. An application generator is a software component that generates a family of applications within a domain. A domain represents a class of applications with similar characteristics and functionality. A micro-language can be seen as a restricted domain-specific language which expresses this variation in natural terms of the domain. A micro-program specifies an instance of the class of applications represented by the micro-language, and a generator is used to automatically generate an implementation of an instance from this specification. Thus, the application generator technology is reused each time a new instance is created.

Our framework uses automatic program transformations to translate a specification, written in the micro-language, into an efficient implementation. The generation process occurs by mapping a micro-program in the micro-language into operations within the abstract machine, which are themselves mapped into an optimized implementation. This mapping process is automated through the use of partial evaluation [6, 10]. Partial evaluation is a program transformation which specializes a program with respect to know values of some of its input.

Domain-specific languages and application generators are both well recognized approaches [3]. They represent a flexible form of reuse that not only allows the reuse of the implementations of abstract functional units, as in the component-based approaches, but also allows the reuse of how these functional units are combined to form a complete system [5]. The only limitation in the amount of design knowledge that can be reused by an application generator is the ability to express that knowledge as an algorithm. Furthermore, application generators allow this knowledge to be reused by non-programmers because the domain-specific language can provide an interface to the domain-user in familiar notations.

Despite these features which make them a good approach to reuse, application generators have limited applicability. The applicability of generator technology is determined by its technical feasibility, as well as development costs. The development costs of generators can be considerably more than the development of an individual application, and must be compared with the long term benefits of reuse. Thus, the critical decision to use generator technology is based on its development costs.

The overall goal of this work is to reduce the effort of generator development. We can summarize our approach as follows. Application generator design is decomposed into two levels, the design of an abstract machine and a micro-language. Each of these can be composed from smaller building blocks to provide further reuse. The complete generation process is achieved through the application of partial evaluation in order to map an application specified in the micro-language to an abstract machine program, and mapping this result to an implementation.

Our work contributes to the design of application generators by providing a structured two-level approach to design, providing a complete path from micro-languages to implementation reusing component-based technology, the uniform use of partial evaluation for automatic program transformation, and the use of a component model to compose and extend application generators.

The following section presents some background on partial evaluation. This is followed by an example that will be used in the following sections, which elaborate

on the two level framework and the application of partial evaluation. The final two sections give a summary of related work and some conclusions.

2 Background

This section presents the background of our approach to program specialization. Program specialization is the process of transforming a general program into one that is more specific, customizing it in some way. In our approach we utilize partial evaluation [6, 10], a well established technique of program specialization. Partial evaluation has been successfully applied in many application areas including operating systems [13], and computer graphics [8].

Partial evaluation is a program transformation which specializes a program with respect to some known inputs by performing as many computations as possible that only depend on the known input. The process of partial evaluation is well suited to each phase of our framework because each phase involves the instantiation of a general component into a specific one.

3 A Working Example

In this section we describe an example that will be used in the following sections to give some concrete context to the framework. The example concerns the development of the control process of a vending machine. Although there are many different kinds of vending machines, for diverse applications, they all have very similar structures. These similarities makes it a suitable application for the use of an application generator. In practice we are applying our approach to real scale applications in collaboration with our industrial partners.

In this example, a vending machine is defined to be a machine that provides a certain service, which is to sell some merchandise selected by the buyer. The vending machine architecture considered consists of a stock unit, an input unit, a coin unit, optionally a display unit, and a central control process. The central control process controls the actions of each unit by sending and receiving messages. The following sections elaborate on the two levels of the proposed framework, and how they are applied to the development of this example generator for vending machine control processes.

4 Abstract Machines

The first level in the design process of our framework is the definition of an abstract machine, which captures the fundamental operations that underly the design of applications within the domain. An abstract machine can be simply defined as a collection of operations which operate on an explicitly defined state. The use of abstract machines is a natural progression from established reuse practices. Starting from the idea of highly parameterized subroutines, one might consider these to be generic components or operations to provide a level of abstraction. This level of abstraction provides insulation between the definition of the operation and an implementation. Given the context of a domain-specific solution, it then seems reasonable that for a given domain, we can define a collection of related operations that cooperate to accomplish some goal. Finally, by introducing an explicit state, we have an abstract machine model. The introduction of an explicit state provides a means of reasoning about the operations and how they interact.

The question to consider now is given an abstract machine how can the operations of the abstract machine be mapped into an efficient implementation? Figure 1 depicts the complete process of application generation in our framework. Parts A and B of the figure correspond to the two levels of the framework, and part B indicates how this mapping from an abstract machine to an efficient implementation takes place. The figure shows that given an abstract machine implementation and an abstract machine program as input to a program specializer, we can generate an implementation. The implementation of the abstract machine is, of course, simply a highly parameterized library. This library can be considered a kind of generic "program", in the sense that it defines the implementation of every possible abstract machine program, whereas the implementation of an abstract machine program is one specific instance of this. Thus, this situation precisely corresponds to what a program specializer does, transform a general program into a more specific one based on some context. In this case, specialization is with respect to the abstract machine program. The result of program specialization yields an efficient program which is optimized with respect to the abstract machine program; it can also optimize inefficiencies introduced by the boundaries between operations.

There are many advantages to this approach. One of these advantages is the opportunity to have many

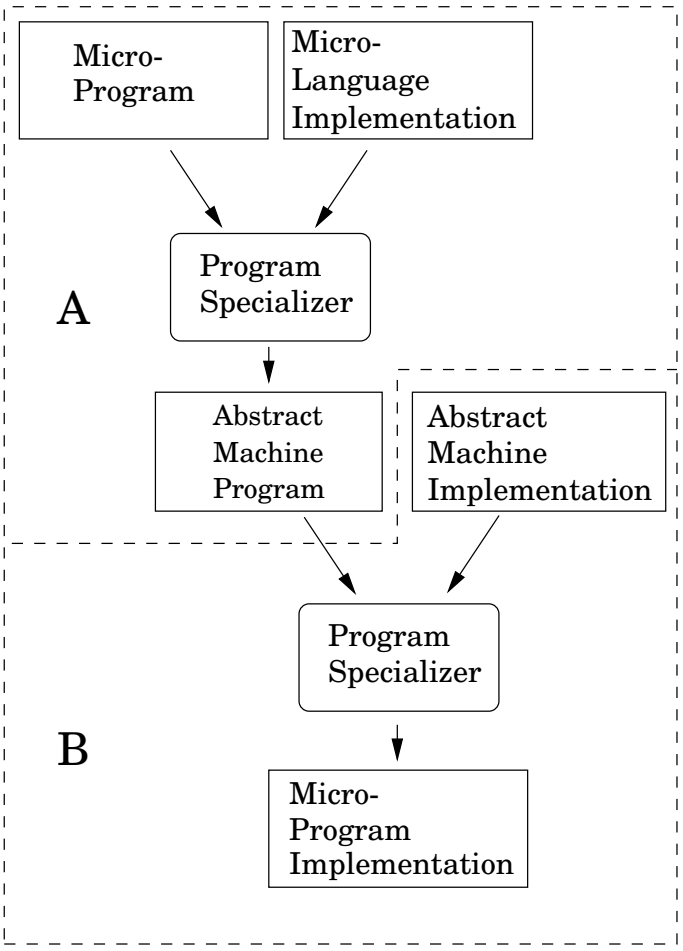

Figure 1: Application generation process

```
State:
  coins :  array of (message, value)
  paid :  value

Operations:
  ...
  -- CHARGE : Accepts coin messages until
  -- the amount requested is paid.
  charge value

  -- GUARD : Executes the list of
  -- operations, if the message has
  -- been received.
  guard message, operation_list
  ...
```

Figure 2: Abstract machine definition fragments.

implementations of a single abstract machine. Each different implementation could also be in a different language. Another benefit is that abstract machines provide a well understood model that can be used to reason about the defined operations. Being able to reason about operations in this way will enable the ability to verify certain properties about micro-programs, or derive other properties like time complexities. The abstract machine model also provides the right level of decomposition to increase reuse of the abstract machine [16].

Figure 2 shows some fragments of an abstract machine for the vending machine example. This definition shows how a state is explicitly defined and what the operations of an abstract machine might be. Notice that while these operations are specific to the domain of vending machine software, they are very general within this domain. The result being that they are suitable for use within any number of generators for the vending machine domain, but are yet restricted enough that it is still possible to reason about them. The second aspect to notice is that the operations are implementation oriented. For example, the guard operator is used to wait for the buyer to make a selection, which is indicated by a message being received by the input unit. This operation does not express "what" is being done, i.e. waiting for a selection, but rather how that is done, wait for a message then execute the operation list. This aspect of the abstract machine is a fundamental difference from the micro-language, which is described in the next section.

5 Micro-languages

The second level of the framework is the definition of a micro-language using fundamental concepts that can modeled by the operations of the abstract machine in the previous level. The key difference between the micro-language and the abstract machine is a micro-program describes *what* an application does and an abstract machine program describes *how* the application operates. Since the micro-language is implemented by the abstract machine operations, the micro-language can be viewed as an interface to the abstract machine. This interface first provides an abstraction to the micro-program designer, and second further restricts the applications that can be expressed to a family of similar applications. Figure 3 shows an example micro-program that specifies a soda machine in the example domain of vending machines. This example

```
machine soda is
 coins dime("coin0")=10
 coins quarter("coin1")=25
 coins dollar("coin2")=100
 stock coke("dispense0")
 stock sprite("dispense1")
 stock oj("dispense2")
 services buy_coke("button0")=
   send "show 75",charge 75, dispense coke;
 services buy_sprite("button1")=
   send "show 75",charge 75,
     dispense sprite;
 services buy_oj("button2")=
   send "show 60",charge 60, dispense oj;
end machine;
```

Figure 3: Vending Machine Micro-program.

```
void service_list()

{
  guard("button0", command_list_1);
  guard("button1", command_list_2);
  guard("button2", command_list_3);
}

void command_list_1()

{
  send_message("show 75");
  charge(75);
  dispense("dispense0");
}
```

Figure 4: Specialized intrepreter fragments.

shows how the micro-language is written to express programs in domain-specific concepts and has a restricted semantics.

As in the previous section, the mapping from a micro-program to an abstract machine program is achieved through the application of partial evaluation This process is depicted in part A of figure 1. The implementation of the micro-language is expressed as an interpreter, which uses the abstract machine operations (by calling the library which implements it). As with the abstract machine, the interpreter can be viewed as generic program that defines the behavior of every possible micro-program. Therefore, partial evaluation is the right tool to use to generate an instance of the interpreter that contains only the behavior of a given micro-program.

Since the abstract machine defines all the computations that are required by applications of the domain and the interpreter uses these operations, then excluding the calls to the abstract machine operations, the computations performed by the interpreter are only concerned with mapping concepts in the micro-program into abstract machine operations. Therefore, all the computation of the interpreter, except calls to the abstract machine operations, will be eliminated by partial evaluation because they depend only on the given micro-program. This requires the designer to ensure that the interpreter and abstract machine are well separated into their proper roles. However, there exists a common analysis in partial evaluation that can readily be used to indicate to the designer if they are not properly separated and indicate where the problems are in the interpreter. Fragments of the result of specialization of an interpreter for the vending machine micro-program in figure 3 is shown in figure 4. In fact, the only remaining computations are calls to the abstract machine operations. The list of services given in figure 3 have been mapped into the service_list procedure which invokes the guard operation for each possible selection, and the result of mapping the action list performed when the buyer selects a coke is the procedure command_list_1.

Another approach to this generation process would be to use compilation technology. The choice to use an interpreter with partial evaluation has two advantages. First, experience shows that compilers are more difficult to design and implement than interpreters. Second, the interpreter approach provides a rapid prototyping environment. The ability of rapid prototyping together with the use of a domain-specific language allow a feedback point early in the generator design process between the micro-language designers and users. Micro-language users can easily understand example micro-programs proposed by language designers and, as well, can propose examples themselves because they express domain concepts.

As there could be many implementations for an abstract machine, the two level framework also provides the possibility to have many micro-languages for one abstract machine. Since the abstract machine can express a wide range applications within the domain and the micro-language only a restricted subset of these, it

is useful to have multiple micro-languages. The use of micro-languages that have a limited semantics has several advantages. The first advantage is the interpreter design is simplified, and a second advantage is micro-programs are simpler and thus easier to reason about. Finally, the restricted semantics allows the possibility to apply automated proof techniques that have been developed for general purpose languages, but have had limited success with general purpose languages due to their generality.

6 Another Dimension of Reuse

This section presents another aspect of our framework which is described in detail in [15]. In addition to providing reuse with application generators, it is also desirable to have methods of reuse for the generators themselves. In order to provide this kind of reuse, both abstract machines and micro-languages can be decomposed into components. An abstract machine component is a collection of operations and their state. A micro-language component defines a collection of language features (e.g., integer expressions) which consists of a definition of the syntax of the language features and functions that can be used to interpret these language features.

We have developed a model that gives the ability to (1) compose components and to (2) extend components. Component extension is performed by adding new operations, and by removing, renaming, extending or replacing existing operations. This model has been successfully applied to the construction of several small, but interesting interpreters.

7 Related Work

One approach to the design of application generators is the use of templates such as that found in [9]. The approach taken in STAGE [5] is to use a metalanguage which, based on the micro-program, generates various code fragments. These techniques provide a lower level view than techniques like DRACO [7], SDDR [2], and AMPHION [11]. These systems require some type of formal specification to build an application generator. DRACO relies on the specification of transformations from one domain-language to another in a hierarchy of languages, where as SDDR uses a formal definition of the domain-language semantics. AMPHION is similar to our framework in that application generation is the generation of subprogram calls to a domain-specific library. However, they use automated theorem proving techniques to derive a series of subprogram calls, and require user provided proof tactics.

GenVoca [1] advocates a methodology of software generator design that consists of assembling domain-specific components, and also uses partial evaluation to optimize between boundaries of these components. In their methodology the components being assembled are software subsystems where our approach is to compose abstract machine operations. In their approach compositions are generated from interconnection like languages. Our work extends this aspect by treating more general domain-specific languages.

8 Conclusion

In this paper we have presented a framework for the development of application generators, and shown how the uniform application of partial evaluation technology can be used to drive the generation process and simplify design. The framework is structured into to levels, an abstract machine, and a micro-language. The design process can be summarized as: First, define an abstract machine which captures the fundamental *operations* that underly the design of applications within the domain. Second, define a micro-language using the fundamental *concepts* that are used to specify an application within a family of applications. Third, implement an interpreter for the micro-language using the fundamental operations defined by the abstract machine. Finally, implement the abstract machine operations.

The resulting generation process is, using partial evaluation, map an application specified in the micro-language to an abstract machine program, and then map this result to an implementation.

The contributions of the framework presented in this paper are as follows:

- A structured approach to generator design.
- A complete path from micro-language to generic components (abstract machine) to implementation.
- The uniform application of partial evaluation which automates the generation process.

References

[1] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.

[2] J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou. Software design for reliability and reuse: A proof-of-concept demonstration. In *Proceeding of TRI-Ada*, pages 396–404, 1994.

[3] J. Bentley. Programming pearls: Little languages. *Comm. of the ACM*, pages 711–716, August 1986.

[4] G. Booch. *Software Components with Ada.* Benjamin Cummings, 1987.

[5] J. Graig Cleaveland. Building application generators. *IEEE Software*, July 1988.

[6] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[7] Peter Freeman. A conceptual analysis of the draco approach to constructing software systems. *IEEE Transactions on Software Engineering*, July 1987.

[8] B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[9] Stan Jarzabek. From reuse library experiences to application generation architectures. In *The Proceedings of the Symposium on Software Reuse*, volume Special Issue of *Software Engineering Notes*, pages 114–122, Seattle, Wasington, April 1995.

[10] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, EngleWood Cliffs, NJ, June 1993.

[11] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *9th Knowledge-Based Software Engineering Conference*, 1994.

[12] G. D. Plotkin. *A Structural Approach To Operational Semantics.* University of Aarhus, Aarhus, Denmark, 1981.

[13] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995.

[14] Marulli Sitariman and Bruce Weide. Component-based software using resolve. *Software Engineering Notes*, 19(4), October 1994.

[15] Scott Thibault and Charles Consel. Micro-language design by composition. In preparation.

[16] Bruce W. Weide and William F. Ogden. Recasting algorithms to encourge reuse. *IEEE Software*, 11(5), September 1994.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex (France)
ISSN 0249-6399